C-to-CoRAM: Compiling Perfect Loop Nests to the Portable CoRAM Abstraction

Gabriel Weisz
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
gweisz@cs.cmu.edu

James C. Hoe
Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, USA
jhoe@ece.cmu.edu

ABSTRACT

This paper presents initial work on developing a C compiler for the CoRAM FPGA computing abstraction. The presented effort focuses on compiling fixed-bound perfect loop nests that operate on large data sets in external DRAM. As required by the CoRAM abstraction, the compiler partitions source code into two separate implementation components: (1) hardware kernel pipelines to be mapped onto the reconfigurable logic fabric; and (2) control threads that express, in a C-like language, the sequencing and coordination of data transfers between the hardware kernels and external DRAM. The compiler performs optimizations to increase parallelism and use DRAM bandwidth efficiently. It can target different FPGA platforms that support the CoRAM abstraction, either natively in a future FPGA or in soft-logic on today's devices. The CoRAM abstraction provides a convenient high-level compilation target to simplify the task of design optimization and system generation. The compiler is evaluated using three test programs (matrix-matrix multiplication, k-nearest neighbor, and 2D convolution) on the Xilinx ML605 and the Altera DE4. Results show that our compiler is able to target the different platforms and effectively exploit their dissimilar capacities and features. Depending on the application, the compiler-generated implementations achieve performance ranging from a factor of 4 slower to a factor of 2 faster relative to hand-designed implementations, as measured on actual hardware.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: REGISTER-TRANSFER-LEVEL IMPLEMENTATION:Automatic synthesis; Optimization

General Terms

Design

Keywords

FPGA computing, High-level Synthesis, Loop optimization, Data reuse

FPGA'13, February 11–13, 2013, Monterey, California, USA.

1. INTRODUCTION

Motivations. Modern FPGAs are capable and efficient computing devices in a wide range of application areas ([9], [20], [11], and [15]), but have failed to achieve widespread use. A major obstacle to the adoption of FPGAs for computing is the high degree of difficulty associated with developing FPGA applications. Prior work such as CoRAM [10], LEAP [1] and VirtualRC [17] have approached this problem from the architectural side by virtualizing an FPGA's external memory and I/O interfaces. Such approaches save development time by avoiding repeated effort in infrastructure development and enabling portability of completed applications. What cannot be solved through architecture and abstraction is the inherent difficulty in manually mapping algorithms to hardware datapaths that target the FPGA's reconfigurable fabric.

Verilog and VHDL, the prevailing design languages for targeting FPGAs, follow a hardware-centric paradigm that requires a designer to directly manage highly concurrent, fine-grained operations with per-cycle coordination. Algorithmic and application experts prefer to operate at a much higher level of abstraction, and would rather use sequential languages such as C. As we will discuss in Section 2, there has been much research and development on compilers that can automatically generate hardware designs for FPGAs from C and other programming languages.

Compiling Perfect Loop Nests for an FPGA. This paper presents work on developing a C compiler that can produce a complete FPGA-based implementation from perfect loop nests with fixed bounds. Perfect loop nests are those in which all computation occurs within the innermost loop body. The innermost loop body can access large data sets residing in off-chip DRAM, using both array and indirect pointer references. This class of programs includes many important scientific and numerical applications. As they typically exhibit a high degree of inherent parallelism and predictable data access patterns, these programs are well suited for implementation on an FPGA. Moreover, others have worked on transforming non-perfect loop nests into perfect loop nests [21].

Unlike previous C-to-gates and C-to-FPGA compilers, our work focuses primarily on achieving efficient use of off-chip DRAM memory bandwidth and on-chip SRAM buffers. We use ROCCC (Riverside Optimizing Compiler for Configurable Computing [23]) to generate streaming hardware kernel pipelines corresponding to the innermost loop body. Our compiler seeks to saturate these kernel pipelines by managing the flow of the input and output data streams be-

tween the kernel pipelines and external DRAM. Our compiler, built on top of LLVM [24], analyzes the memory references in a loop nest for dependencies and access patterns. The compiler then introduces optimizations to increase parallelism, coalesce memory accesses, infer data reuse, and support efficiently strided memory accesses.

CoRAM Compilation Target. Unlike previous C-to-FPGA compilers, our compiler does not directly target the bare FPGA fabric. Instead, we target the CoRAM FPGA computing abstraction. Figure 1 presents a conceptual depiction that demonstates how the CoRAM abstraction enforces a separation of concerns between processing and data movements. Under the CoRAM abstraction, applications are partitioned into kernel pipelines, which implement computation, and control threads, which sequence control flow and data transfers. A kernel pipeline performs a specific task each time it is invoked, interacting solely with attached CoRAM SRAM buffers and control threads. Section 3 discusses the CoRAM abstraction in more detail.

Targeting the virtualized CoRAM abstraction allows our compiler to create implementations for any supported FPGA platform, either natively in future FPGAs or using soft-logic on today's devices. In this paper, we tested our compiler using the Xilinx ML605 and Altera DE4 platforms. On these platforms, the CoRAM abstraction is supported by a soft-logic microarchitectural implementation. The CoRAM abstraction greatly simplifies our compilation task because control flow and the management and optimization of memory data movements are easily expressible in CoRAM control threads. The CoRAM abstraction also serves to mask even non-trivial platform differences, such as the number of DDR memory channels. Our evaluation shows that our compiler can not only transparently support different FPGA platforms, but can also effectively optimize our test programs (matrix-matrix multiplication, k-nearest neighbor, and 2D convolution) for the characteristics of the different targets. On these test programs, our compiler produced implementations comparable to hand-designed implementations in performance, ranging from a factor of 4 slower to a factor of 2 faster.

Overview. The remainder of this paper is organized as follows. Sections 2 and 3 provide background on prior art in C-to-hardware compilation and the CoRAM abstraction. Section 4 presents the design and implementation of our compiler. Section 5 presents our evaluations. Finally, section 6 concludes and identifies future extensions.

2. RELATED WORK

2.1 C-to-Hardware Compilers

There is extensive prior and current work, both commercial and academic, on compiling from high-level software programming languages to hardware implementations. We provide a brief survey of the most closely related projects in the field of C-to-hardware compilation.

ROCCC [23], Impulse-C [25], Handel-C [26] and Catapult-C [27] are all examples of C-to-hardware compilers for generating hardware kernels. Loop nests are among the key program structures exploited by these tools for parallelism and performance. A common feature is that users can annotate loops for pipelining or unrolling optimizations. These tools do not provide integrated end-to-end support for off-chip DRAM access; the generated kernels interface with on-chip

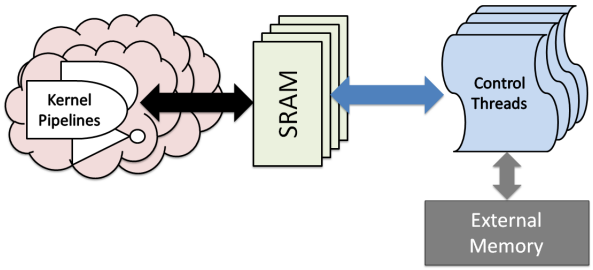

Figure 1: CoRAM Abstraction Concept.

SRAMs or assume a streaming data interface that relies on external logic to provide input data and handle output data. Our work uses ROCCC to generate streaming pipelined implementations of hardware kernels from the innermost loop body of the loop nest. Xilinx Vivado [28] (formerly AutoESL) is a Xilinx-specific tool that compiles C code to hardware kernels that can utilize on-chip BRAMs for interfaces.

Compared to the aforementioned work, the applicability of our compiler is limited to perfect loop nests. However, our compiler produces complete, optimized implementations that can interact with large data sets residing in DRAM. Altera C2H [29] does produce accelerators with pipelined memory accesses as a part of the Altera NIOS framework, but does not automatically perform optimizations to exploit data reuse and/or interchange loops to increase parallelism.

LegUp [8] represents a different class of hardware compilers for building hybrid hardware/software embedded systems. Their focus is on starting with a software-only implementation, and reaching greater performance and/or power/energy efficiency through the addition of hardware accelerators of hot program sections.

Our example programs are implemented from C source code (Listings 2, 3, 4, and 5). It would be possible to use existing C-to-gates tools such as the ones described above to implement them on an FPGA, but some work would be required (through the introduction of source code annotations at the very least) in order to create high performance parallel implementations that implement reuse-optimized data transfers between external DRAM and block memories.

2.2 Loop Nest Optimizations

Loop Nest Optimizations are also well studied in the literature. Polyhedral analysis is the state-of-the-art technology for automatic pipelining and parallelization of loop nests [18]. It defines an *iteration vector* that encodes how variables change within the loop nest. Optimizations are constructed as affine transformations of these iteration vectors [5].

Diniz and Park [14] describe compiler analyses using the polyhedral model to find data reuse and reordering. Alias, Pasca, and Plesco [2] use polyhedral analysis to tile and parallelize applications for implementation on an FPGA. Bayliss and Constantinides [6] use the polyhedral model to optimize the memory accesses of an application and create hardware for address generation. Cong, et al. [13] investigated a broad set of mechanisms, including loop interchange, for optimizing data reuse, buffer sizes, and memory bandwidth.

The papers referenced above have a strong focus on specific optimizations. The goal of our work is not to contribute new loop nest optimizations but to employ them—extensively leveraging the analysis infrastructure built into

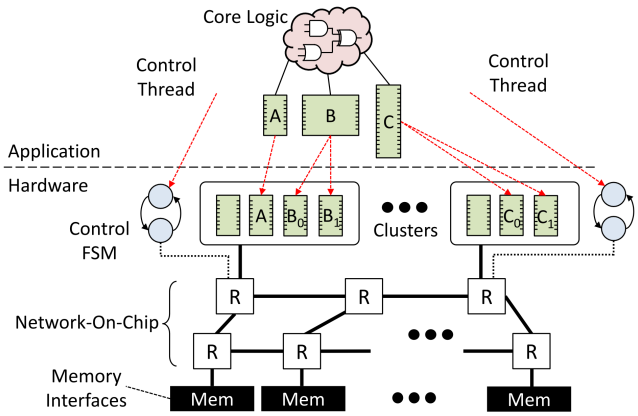

Figure 2: CoRAM Microarchitectural Sketch (figure from [12]).

LLVM [24]—in order to efficiently support C-to-hardware compilation for the CoRAM abstraction [10]. Our current capabilities (presented in Section 4) do not require polyhedral analysis. Extensions to incorporate polyhedral analysis would allow us to automatically introduce memory blocking transformations and set block sizes. The test programs used in this paper (Listings 2, 3, 4, and 5) are explicitly blocked by hand at the source code level. A polyhedral framework, such as LLVM's Polly [16] project, provides a path to incorporating these analyses and optimizations.

3. CORAM TARGET ABSTRACTION

Our compiler targets the CoRAM abstraction, which provides a convenient virtualization of the communication and control infrastructure that connects in-fabric kernel pipelines to external memory (DRAM). Figure 1 offers a conceptual depiction of the CoRAM abstraction. Kernel pipelines are localized in space and in time, and perform a specific task each time they are invoked, interacting solely with attached SRAM buffers and control threads.

The application developer uses a C-based language (with pointer support) to define a set of control threads that dynamically manage the contents of the SRAMs and coordinate the invocation of the kernel pipelines. This separation of concerns between processing and data movement allows for a high-level virtualized execution environment that simplifies an application's development and improves the application's portability and scalability.

A fully CoRAM-compliant FPGA would implement the underlying mechanisms for data transport between kernel pipelines and external memory with native, hardwired datapaths. This abstraction layer, akin to an ISA for processors, enables application-level compatibility across different FPGA families and device generations.

Figure 2 offers a microarchitectural sketch of a possible datapath implementing the CoRAM abstraction. This design can be scaled to up to thousands of CoRAM clusters, depending on the capacity of the FPGA [12]. CoRAM SRAM blocks, like embedded SRAMs in modern FPGAs, are arranged into columns and organized into clusters. A cluster is a group of CoRAM blocks attached to the same network-on-chip endpoint. Grouping blocks together into clusters reduces the number of network endpoints needed, and provides a mechanism to compose several CoRAMs into larger blocks (with a customizable aspect ratio), but limits available bandwidth due to the shared network endpoint. Each cluster is managed by an attached Control Unit, which is responsible for executing the control threads that run within the cluster. Control threads in the CoRAM programming abstraction can be realized by direct synthesis into reconfigurable logic as finite state machines, or can be executed on dedicated microcontrollers. In addition to native CoRAM constructs, a customizable library of "personality" extensions, layered on top of the basic CoRAM SRAM blocks and APIs, implement FIFO queues, caches, and similar structures.

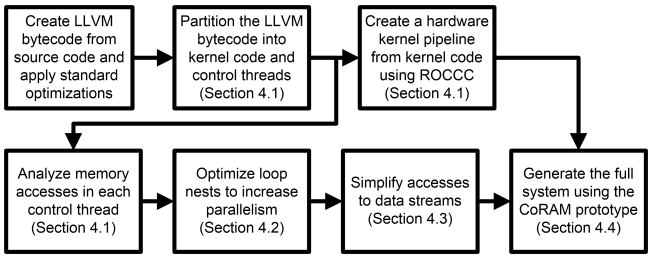

Figure 3: Generating an FPGA system from source code.

While no FPGA available today natively supports the CoRAM abstraction, it is available for off-the-shelf Xilinx and Altera FPGA platforms (ML605 and DE4, respectively), supported by soft-logic implementations of the necessary mechanisms [22]. The implementations include a network-on-chip that uses CONNECT [19], control threads that are compiled into state machines, and CoRAM blocks that are mapped on top of conventional block SRAMs. More details are available in our prior work [10] and publicly available prototype [22].

4. COMPILER IMPLEMENTATION

Our current compiler can only handle perfect loop nests with fixed loop bounds. Although limited, perfect loop nests appear extensively in scientific and numerical kernels, and others have worked on transforming non-perfect loop nests into perfect loop nests [21].

Our compiler is implemented as a compiler pass for the LLVM compiler infrastructure [24]. It operates on LLVM bytecode instructions using built-in facilities for program analyses and manipulation. The programmer is presented with the familiar "make" build process that automates the flow from C source code to a complete FPGA design.

Figure 3 shows the steps that create an FPGA system from software source code. These steps are:

- Creating LLVM bytecode from source code (and applying standard optimizations).
- Partitioning the LLVM bytecode into kernel code and control threads (Section 4.1).
- Applying ROCCC to kernel code in order to generate kernel pipelines (Section 4.1).
- Analyzing memory accesses to discover data access patterns (Section 4.1).
- Optimizing control thread loop nests (Section 4.2).
- Simplifying data streams (Section 4.3).
- Producing the complete system using the CoRAM Design Generator (Section 4.4).

We continue with a detailed discussion of the important steps in the implementation process.

```
1 Returns True if the instruction is a
    kernel instruction; otherwise False
2 Bool IsKernel(Instruction i)
3  If  i is a store
4   return True
5  Else if i is a non-loop branch
6   return True
7  Else if i is a loop branch
8   return False
9  Else
10   For each Instruction u that consumes
       the value produced by i
11    If u uses i as a load/store address OR
12      IsKernel(u) == False
13     return False
14   return True          // catch all
```

Listing 1: Classification Algorithm.

4.1 Kernel Extraction and Synthesis

The CoRAM abstraction requires a separation between the processing kernel – the portion of the application actually performing computation – and the code implementing data transfers and control flow. This separation is achieved by classifying bytecode instructions as computing or non-computing bytecode instructions and extracting the computing bytecode instructions.

Listing 1 gives simplified pseudo code of the decision function IsKernel( ) that decides whether or not each of the bytecode instructions in the innermost loop body belongs to the processing kernel. In essence, a bytecode instruction is a part of the processing kernel only if its entire subtree of dependent bytecode instructions all belong to the processing kernel. Caching the categorization of previously processed instructions ensures that each bytecode instruction is only visited once; the runtime is therefore linear in the number of bytecode instructions.

The bytecode instructions that are flagged as belonging to the processing kernel are re-emitted as a C function. Load and store bytecode instructions in the emitted function are used as placeholders for creating input and output queues during hardware mapping. The addresses used by these bytecode instructions, retained in the original loop nest, are used in later optimization phases. The compiler can recognize memory addresses that are used as accumulation variables (first read and then written to in the loop body) and pair them for special processing. Separate control threads are created for each input and output variable, which simplifies later optimizations.

The processing kernel is compiled by ROCCC [23] into a hardware kernel pipeline, with streaming input and output interfaces replacing load/store bytecode instructions. Each input and output streaming buffer is mapped to distinct CoRAM memory blocks and managed by a different control thread. In Section 4.2 we will explain optimizations that increase processing throughput by instantiating multiple concurrent kernel pipelines and/or increasing the size of kernel pipeline through unrolling.

Figure 4 offers a generic system containing multiple parallel kernel pipelines that receive data from CoRAM FIFO personalities that implement input streaming buffers (A and B). The CoRAM personalities in the figure have been composed with additional logic to create "Reuse Buffers" that implement a data reuse pattern detected by the compiler

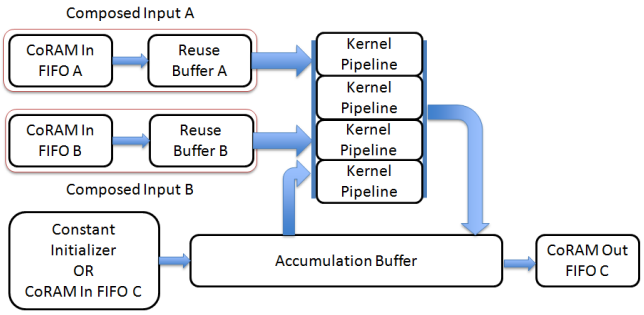

Figure 4: Block Diagram of Generic Streaming CoRAM Kernel Pipelines with associated CoRAM Personalities and Reuse Buffers.

(discussed in Section 4.3). Bandwidth requirements are also reduced, as after recognizing an accumulation variable, the compiler has introduced an Accumulation Buffer that removes the need for explicit synchronization between read and write threads, and an Initializer to avoid reading data initialized by a constant from DRAM (C).

The compiler produces optimized control threads to fill the input buffers as quickly as possible. Without data-dependence stalls, each hardware kernel pipeline can issue one loop iteration every cycle. The outputs of an iteration are produced in-order after the pipeline delay. The control threads must obey data dependencies and will stall an input data stream if it depends on the output of in-flight iterations.

LLVM's Scalar Evolution framework is used to analyze the memory accesses in loop in each control thread, to infer information that is used in later optimizations. Each variable used as an address is analyzed, and its progression at each loop is flagged as exhibiting one of the following patterns:

- Unchanging: same address across iterations.
- Stride-1: consecutive data items across iterations.
- Computable: non-consecutive data items following an analyzable progression, such as stride-permutation and indirect accesses.
- Undetermined progression.

4.2 Loop Optimizations

The ROCCC generated kernel pipelines can only be saturated if there are no data dependencies across iterations of the innermost loop. When the kernel pipeline is replicated for higher performance, even greater parallelism is required to achieve peak computational throughput. We apply one of two optimizations to obtain the necessary parallelism.

Loop Interchange. The default mechanism used for increasing parallelism is interchanging loops. This is a classic loop nest transformation. We move loops at which an accumulation occurs outwards in the loop nest. Selecting an appropriate loop ordering requires a balance between creating potential parallelism and minimizing on chip buffering—too little parallelism causes pipeline stalls; too much parallelism may require excessively large on-chip buffers.

Our compiler follows a simple heuristic for reordering that works especially well when the loop nest has been blocked for better memory locality. We select one or more consecutive loops at which the output variable is accumulated, and move these loops outward until (1) encountering another accumulating loop; (2) reaching the top of the loop nest; or (3) reaching a user-specified threshold in the amount of output data that must be buffered. The intent of this strategy

is that the user may match the CoRAM block size to the application block size.

Reordering loops without taking into account data patterns might convert sequential memory accesses into strided accesses. Section 4.3 will show how our compiler can transpose strided data in transit, allowing these accesses to be supported with little performance penalty.

Loop Unrolling. The user may choose to fully unroll the innermost loop instead of interchanging loops. Unrolling the innermost loop creates a new function that cascades instances of the baseline processing kernel. As the entire loop has been unrolled, it no longer can contain a loop-carried dependency, although the resulting hardware may contain an input for an accumulation at another loop. The cascaded kernel pipeline created by ROCCC reads each input, with the exception of the accumulated variable, for each instance of the baseline kernel pipeline. The accumulated variable is read once and passed between the cascaded instances. Our compiler can optionally attempt to balance the unrolled function's dataflow graph in order to minimize long paths (and the corresponding pipeline depth). Balancing may be disabled if the order of the operations must be preserved, such as when implementing certain floating-point kernels.

Trade-off. The decision to unroll or interchange loops is currently left to the user. This decision affects both the amount of data that is buffered and the organization of the streaming data buffers. An implementation that interchanges loops will likely implement a larger number of discrete kernel pipelines (with a shorter pipeline depth) than one that unrolls the innermost loop. This reduces the required number of pipeline buffers, but increases the number of kernel pipelines and accumulated values (one per kernel pipeline) that must be buffered. Different realizations of matrix-matrix multiplication are presented in Section 5.1.1, where our results show that loop interchange produces a smaller design. In contrast, Section 5.2 presents code for which loop unrolling produces a smaller design.

4.3 Simplifying Data Streams

The compiler next optimizes the loop nest of each control thread to coalesce memory accesses, infer data reuse, and simplify the loop nest. Each kernel pipeline receives the same data that it would receive without optimization, allowing each control thread to be processed independently.

Figure 5 illustrates four transformations from the perspective of an input data stream. Output data is optimized similarly. The patterns are inferred from information gathered earlier in the workflow (See Figure 3), and may span multiple loops. In each illustration, the top portion shows a pattern that is repeated in the original full data stream (flowing left to right). Elements of the data stream are labeled with their addresses. The bottom portion shows the reduced data stream as actually fetched from external DRAM, and how the original data stream is recreated. The various patterns are processed as follows:

(a) Repeat: The addressing pattern comprises repeated memory reads of the same address (A), either over the innermost loop or multiple consecutive inner loop levels. The control thread's loop is replaced by a single memory read request, and a buffer reproduces the original stream. In addition to repeating a single address, any of the following patterns may be repeated to reuse larger blocks.

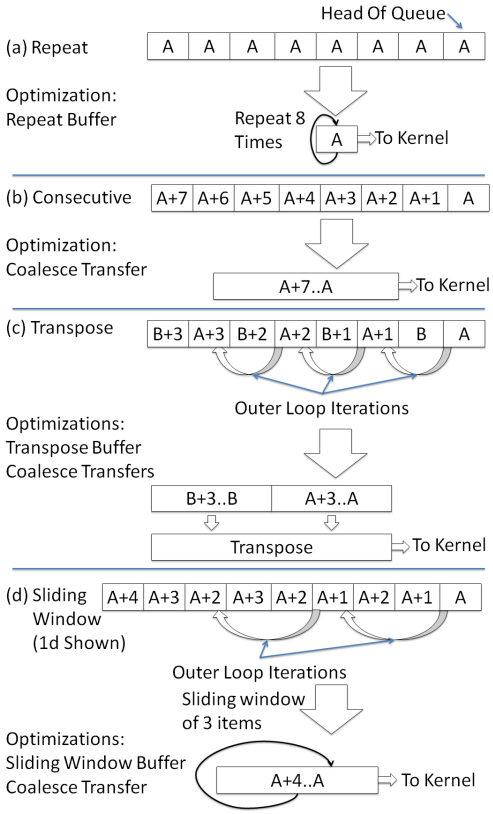

Figure 5: Access patterns and associated optimizations.

(b) Consecutive: The addressing pattern contains n stride-1 accesses of data items starting at A, such as when visiting successive elements of an array. The loop for generating the n consecutive reads is replaced by a single burst read request of size n. The underlying CoRAM-based streaming buffer is not altered as the same elements pass through it.

(c) Transpose: The addressing pattern corresponds to a strided memory read loop, such as when visiting the elements of a sub-block of a row-major 2D array in column order. The control threads generate burst read requests for required rows of the sub-block and buffer the entire sub-block. A stride-permutation personality reorders the stream on the fly to recreate the original strided data stream. The control thread's loop nest is reduced by implementing burst reads of entire sub-block rows. Strides need not be fixed, and may depend on indirect accesses (as in Listing 3).

(d) Sliding Window: The pattern corresponds to n successive reads to consecutive addresses starting at A, followed by another n successive reads from consecutive addresses starting at A+1, and so on. This "sliding window" pattern is common in filters. The control thread generates a simple burst read from consecutive memory locations starting at A. A special CoRAM personality buffers all of the data in the sliding window in block memory, and ensures that the data is delivered in the correct order. As in (b) above, inner loop nests for generating this complex pattern are completely simplified away with a single burst read request. The figure presents a one dimensional sliding window, but the compiler also supports two dimensional sliding windows. Data items within a row are routed to parallel kernel pipelines by multiplexers and interleaved across rows.

Platform	Xilinx ML605	Terasic DE4
FPGA	Xilinx LX240T [30]	Altera EP4SGX530[31]
Logic Cells (Overhead (%))	241,152 33%	531,200 30 %
Block Memory (Overhead (%))	14,976 KB 13%	27,376 KB 18%
DSPs	768	1,024
DRAM Bandwidth	6.4 GB/s	12.8 GB/s
DRAM Capacity	512 MB	2 GB
PCI Express	x8	x8
UART Speed	500 KBits/s	115 KBits/s

Table 1: Parameters for test platforms. "Overhead" includes the memory controllers, network-on-chip, among other components.

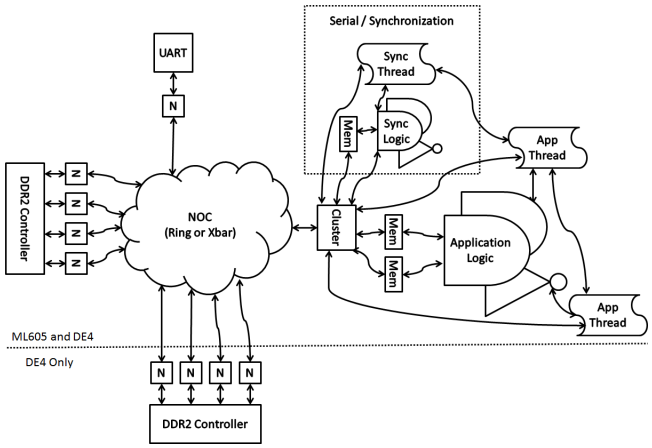

Figure 6: Complete design resulting from compilations.

4.4 FPGA Design Generation

The compiler backend produces the FPGA implementation by invoking the CoRAM Design Generator with: (1) the ROCCC-generated VHDL kernel pipelines; (2) The control threads; and (3) the required CoRAM personalities. Basic CoRAM personalities such as streaming data buffers already exist in the CoRAM personality library. A one-time effort was necessary to build reuse buffer CoRAM personalities that support repeated data, transposed data, and sliding windows.

The CoRAM Design Generator creates a single FPGA design containing: (1) the kernel pipelines; (2) the FSMs compiled from the control threads; and (3) the soft-logic implementation of the CoRAM architecture (including the underlying platform-specific on-chip communication, external memory modules, and I/O modules). This FPGA design is processed by platform-specific synthesis tools that create a bitstream for programming onto a device. The CoRAM Design Generator also generates RTL models for simulation-only studies that allows flexibly specifying the parameters of hypothetical FPGA target platforms (see Section 5.1.4).

The CoRAM Design Generator currently supports the Xilinx ML605 and Altera DE4 platforms. Table 1 summarizes the most salient characteristics of the two platforms. The two rows in Table 1 labeled "Overhead" indicate resources consumed by the soft-logic CoRAM implementation without an attached user design. This is not a pure overhead as, even in hand-crafted designs, kernel pipelines must be supported by non zero-cost infrastructure for communicating with DRAM and I/O.

Figure 6 offers a block diagram of the complete FPGA design. Most of the details shown in the figure, aside from control threads, hardware kernels, and CoRAM personalities, are below the CoRAM abstraction and handled by the CoRAM Design Generator.

The figure shows memory controllers connected to a soft-logic network-on-chip through multiple network endpoints (labeled "N"), which avoids bottlenecking the available memory bandwidth. The network-on-chip can (interchangeably) implement a variety of topologies, including the ring and crossbar networks used in the evaluation, which vary in logic cost and provided bandwidth. On the Altera DE4 with two memory controllers, the global address space is interleaved between the two channels at 256-byte boundaries. We elect to use a single CoRAM cluster to service all of user logic generated by our compiler. The cluster connects multiple CoRAM blocks and the associated control threads to the network-on-chip—one block and thread per memory variable in the innermost loop body. Communication channels between the control threads and the hardware kernel pipelines are automatically inserted to support synchronization, including a staging mechanism (labeled "Serial/Synchronization"), which allows the host computer to communicate with the FPGA platform for data transfers and to collect performance results.

Switching between ML605 or DE4 targets is trivial - a configuration file specifies the number of memory controllers (and associated network endpoints) to instantiate, and the CoRAM Design Generator provides wrappers for floating point IP cores and scripts to invoke device specific bitstream generation tools.

5. EVALUATIONS

We evaluate our compiler on three code examples: single precision matrix-matrix multiply, k-nearest neighbor, and two dimensional single precision convolution (actual source code included in Listings 2, 3, 4, and 5). We use standard "blocked" implementations of these algorithms. For example, in Listing 2, the 3 outer loops correspond to the familiar triple-loop in standard matrix-matrix multiplication, except in a blocked implementation. Multiply-accumulate operations are performed on sub-matrix tiles (thus the inner 3 loops) and not on individual elements. By buffering and operating on the data tiles on chip, blocking algorithms increase the number of times a data value fetched from memory is reused.[1] All experiments run kernel pipelines, reuse buffers, and other components at 100MHz, the highest clock speed currently supported by the CoRAM prototype.

5.1 Matrix-Matrix Multiply

Listing 2 gives the source code for matrix-matrix multiplication (MMM) used to evaluate our compiler. In order to work with the current compiler, the matrix sizes (separate SIZE_I, SIZE_J, SIZE_K to allow for non-square matrices), and block sizes (separate BI, BJ, BK to allow for non-square blocking) must be fixed at compile-time. SIZE_I, SIZE_J, SIZE_K must be multiples of BI, BJ, BK, respectively.

The innermost loop (line 8) in this baseline MMM implementation has a loop-carried dependency through the accumulation of C[i*SIZE_I+j], preventing parallel or overlapped

[1]While not a part of the current work, polyhedral techniques[7] could automatically create blocking structures.

```
1 void mmm(float *A,float *B,float *C,
    unsigned SIZE_I,unsigned SIZE_J,
    unsigned SIZE_K) {
2   unsigned i,j,k,i0,j0,k0;
3   for(i0=0;i0<SIZE_I;i0+=BI)
4    for(j0=0;j0<SIZE_J;j0+=BJ)
5     for(k0=0;k0<SIZE_K;k0+=BK)
6      for(i=i0;i<i0+BI;i++)
7       for(j=j0;j<j0+BJ;j++)
8        for(k=k0;k<k0+BK;k++)
9         C[i*SIZE_I+j]+=A[i*SIZE_I+k] *
            B[k*SIZE_K+j];
```

Listing 2: Blocked MMM Source Code.

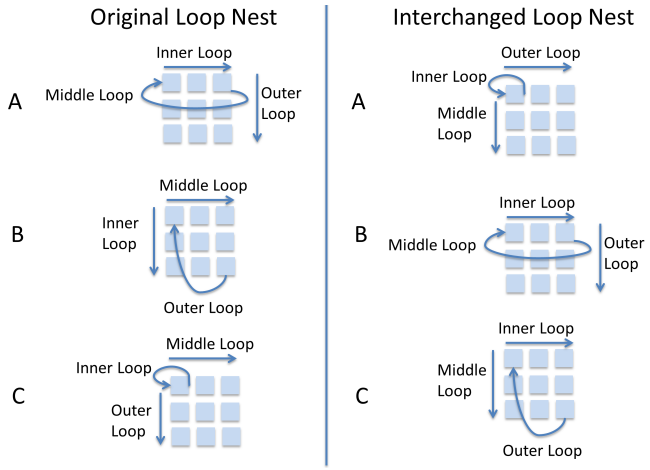

Figure 7: MMM variable progressions.

executions of the loop body. As discussed in Section 4.2, either loop interchange or loop unrolling must be applied to achieve hardware concurrency. Figure 7 examines the data access patterns of A, B, and C within the innermost three loops, for both before and after the loop interchange. In the bottom left, which presents the accesses pattern of the accumulation variable C, the figure shows that the same matrix element is read and written by all iterations of the innermost loop, creating a loop-carried dependency. After interchange, the invariant access is now at the outer loop, and the inner and the middle loop iterations are no longer data dependent, and can be executed in parallel or be pipelined.

In experiments that implement parallel kernel pipelines, each data stream is distributed "round robin" among them, and Reuse Buffers in front of the kernel pipelines allow them to issue a computation each cycle. The arrangement of kernel pipelines and Reuse Buffers as presented in Figure 4.

The inner loop at line 8 in the original code may be unrolled rather than interchanged via a user specified compiler flag. For experiments in Section 5.1.1 that implement loop unrolling, the data access patterns that follow those on the left side of Figure 7. The middle and outer loops provide the data independent iterations to the pipelined kernel based on the fully unrolled inner loop. The inner loop's loop-carried dependencies caused by the memory variable C become signals that pass from pipeline stage to pipeline stage; only the final accumulated value of C from the final stage is written to memory. The arrangement of kernel pipelines and reuse buffers is once again as depicted in Figure 4, except that there is only one kernel pipeline that accepts multiple data items from each input data stream. There is a singular accumulated value for each computation that feeds back into the kernel pipeline across blocks.

#	Opt	BI,BJ	%LC/MEM/DSP	GFlop/sec(%)
1	I	64, 64	84 / 32 / 41	12.6 (98)
2	U	64, 64	83 / 92 / 41	12.6 (98)
3	I	32, 128	82 / 32 / 41	10.4 (81)
4	U	32, 128	83 / 92 / 41	10.4 (81)
5	I	32, 64	83 / 32 / 41	8.6 (67)
6	U	32, 64	81 / 92 / 41	8.6 (67)
7	I	128, 64	86 / 33 / 41	12.7 (99)
8	U	128, 64	86 / 92 / 41	12.6 (98)

Table 2: ML605 Block size experiments configurations. For all trials, square matrices of size 1024 are used. For the Optimization column, "I"=Interchange and "U"=Unroll.

5.1.1 Block Sizing and Loop Optimizations

The first experiments examine the effects of different block sizes in combination with different loop optimizations. Our goal is to demonstrate the range of possible outcomes enabled by the rapid exploration of design options through our compiler. We use a block size of 64-by-64 as the baseline on square 1024-by-1024 MMM. We then consider non-square blocks that are twice, half and the same size as the baseline block size. For each choice of block size, we apply either the loop interchange or the loop unrolling optimization, and create kernel pipelines containing 64 single precision floating-point adders and 64 single precision floating-point multipliers. When synthesized for the LX240T on the Xilinx ML605, ROCCC-generated pipelines at 100MHz yield a theoretical peak performance of 12.8 GFlop/sec.

Table 2 reports the measured performance on the Xilinx ML605 for all combinations of 4 block sizes (indicated by columns **BI** and **BJ**) and the 2 loop optimizations (indicated by column **Opt**: Interchange or Unrolling). The fourth column indicates resource utilization in terms of the percentage of the LX240T's Logic Cells (**LC**), Block Ram (**MEM**), and DSP Blocks (**DSP**) used by each design (as reported by ISE 13.4). Execution times are measured using a hardware counter that counts clock cycles from when computation has started (input arrays A and B starting in DRAM) to when computation has finished (output array C completely written to DRAM). The column **GFlop/sec** reports the absolute performance in GFlop/sec and the percentage of the 12.8 GFlop/sec peak attained.

Rows **1** and **2** show that square blocks are effective, as both experiments achieve over 98% of peak performance; the loop interchanging optimization results in less BRAM usage. The results in the remaining rows show that poor choices of block sizes can indeed negatively impact performance. Rows **3** and **4** show that a rectangular 32-by-128 blocking (the same size as the baseline square 64-by-64 blocking) leads to a drop in performance (only 81% of the peak performance). Halving the block size to 32-by-64 (rows **5** and **6**) further hurts performance even more (down to 67% of peak performance). Doubling the block size to 64-by-128 (rows **7** and **8**) provides minimal performance improvements.

As a point of reference, the best performing implementation of MMM on the Xilinx ML605 we could find was Bao, et al. [4]. Their hand-tuned design uses a systolic array of 2x2 multiply-accumulators connected by a ring network, and implements 32 processing elements running at 200 MHz. They achieve 50.4 GFlop/sec on MMM of size

```
1 void mmm_indirect(unsigned **base,
   unsigned SIZE_I,unsigned SIZE_J,
   unsigned SIZE_K) {
2  unsigned i,j,k,i0,j0,k0;
3  float **A=(float **)base[0];
4  float **B=(float **)base[1];
5  float **C=(float **)base[2];
6  for(i0=0;i0<SIZE_I;i0+=BS_I)
7   for(j0=0;j0<SIZE_J;j0+=BS_J)
8    for(k0=0;k0<SIZE_K;k0+=BS_K)
9     for(i=i0;i<i0+BS_I;i++)
10     for(j=j0;j<j0+BS_J;j++)
11      for(k=k0;k<k0+BS_K;k++)
12       C[i][j]+=A[i][k]*B[k][j];
```

Listing 3: Blocked MMM with Indirection.

#	Kern	Net	% LC/MEM/DSP	GFlop/sec
9	64	Ring	97 / 28 / 25	12.6
10	128	Xbar	96 / 38 / 50	25.4

Table 3: MMM Experiments on the DE4. In these trials, square matrices of size 1024 and blocks sized to the number of kernels.

2048-by-2048.[2] Their carefully hand-tuned MMM is 4 times faster (2x from higher frequency and 2x from better logic packing). However, our design is automatically produced from C source code by a compiler that can handle perfect loop nests and can flexibly re-target any CoRAM supported platform.

5.1.2 Indirection

Support for indirect memory references is evaluated using another common matrix-matrix multiplication implementation (Listing 3) with indirect row-major 2D matrices. We also load the base address of each matrix from memory.

The hardware generated by this program has the same basic structure as the one generated for Listings 2, since the real differences manifest mostly in the control threads. The implementation of this program does require an instance of the CoRAM Load-Store personality for each control thread, which allows the control threads to directly access DRAM in order to dereference pointers. In the other examples, control threads only make requests to transfer data between DRAM and the streaming buffers, and do not examine the memory data values themselves. The Load-Store personality supports memory reads and writes by the control threads, and includes a scratchpad for caching dereferenced values.

In general, memory indirections can introduce a significant performance overhead. In this particular example, the indirections are infrequent and readily amortized with the help of the scratchpad. We achieved 98% of the peak performance of the kernel pipelines on the Xilinx ML605, using the same configuration as Row **1** of Table 2.

5.1.3 Network and Memory Controllers

We recompiled the unmodified MMM program in Listing 2 for the Altera DE4 platform to test our claims of portability and scalability. The EP4SGX530 FPGA on the Altera DE4 provides twice the logic capacity as the LX240T on the ML605, and twice the DRAM bandwidth through two independent DDR controllers. These differences are transparent to the programmer, who requests two memory controllers from the CoRAM Design Generator ("NUM_MC=2") and invokes a (provided) DE4 bitstream generation script.

Row **9** in Table 3 reports the synthesis and performance results achieved by simply recompiling the best-performing configuration on the Xilinx ML605 (Row **1** of Table 2, 64-by-64 block, 64 parallel kernel pipelines). The resulting design reached the same performance as on the Xilinx ML605, approximately 98% of the theoretical peak performance of the kernel pipeline (also synthesized to 100 MHz on the DE4).

With the extra capacity of the DE4's EP4SGX530, we can instantiate a larger number (128) of parallel kernel pipelines than on the Xilinx ML605, and do so by setting "MAXKERNELS=128" and similarly redefining the constant that sets the block size. Through the CoRAM Design Generator, we also selected the higher-performance crossbar network-on-chip topology to deliver DE4's dual DDR memory bandwidth and service the larger number of kernels. As row **10** in Table 3 reports, we can easily double performance by retuning to the greater capacity of the Altera DE4 platform.

Parameter	Kintex-7 XC7K70T [32]	Virtex-7 XC7V2000T[32]
Logic Cells	65,600	1,954,560
Block Ram	4,860KB	46,512KB
DSP	240	2,160
Mem Controllers	1	4
Kernels Used	8	512
Data Size	128	1024
Block Size	16	512
GFlop/sec	1.44	93.6

Table 4: Simulations targeting Xilinx 7-series chips.

5.1.4 Scaling

To further demonstrate portability and scalability, we compile the MMM code for hypothetical FPGA platforms sized to mimic the smallest and largest of Xilinx's Virtex-7 parts. We conservatively assume the kernel pipelines run at the same 100 MHz as in the earlier experiments. The assumed FPGA configuration and the performance results are shown in Table 4. The results show that our compiler is able to produce designs to take up the available level of logic resources of the two FPGAs and produce a commensurate level of performance for the logic resources consumed. In the small extreme (which mimics the small Kintex FPGA), we found the overhead of the CoRAM infrastructure to be nearly $\frac{2}{3}$ of the available logic resources,[3] and we only fit 8 kernels. In the large extreme, the compiler scaled up the design and performance as expected. We assume that the large FPGA will be fitted with four DDR channels.

5.2 k-Nearest Neighbor

The second example in our evaluation is the k-Nearest Neighbor (k-NN) algorithm (Listing 4). This algorithm finds the k best matches for each input vector (called a *target descriptor*) among a larger set of *reference descriptors*. We use the square of Euclidean distance metric to compare 128 element vectors of single byte integers.

In the example code, we block the iteration over target descriptors into two loops (lines 2 and 4) to increase data

[2] As both designs overlap computation with communication, and we report throughput rather than absolute computation time, the difference in matrix sizes is irrelevant.

[3] As previously discussed, this is not pure overhead, as even hand-crafted designs require infrastructure for buffering and communication.

```
1 void match(unsigned char *target,
    unsigned char *ref, void *out,
    unsigned Sz, unsigned TSz,
    unsigned RSz, unsigned Ni) {
2  for (unsigned bi=0;bi<TSz;bi+=BSz) {
3   for(unsigned ri=0;ri<RSz;ri++) {
4    for(unsigned li=bi;li<bi+BSz;li++) {
5     unsigned cur=0;
6     for(unsigned i=0;i<Sz;i++) {
7      short val=(short)target[li*Sz+i]-
8       (short)ref[ri*Sz+i];
9      unsigned short v=val*val;
10     cur+=v;
11     }
12     StoreMin(out,cur,Ni,RSz,BSz,TSz)[4];
```

Listing 4: k-Nearest Neighbor Source Code.

#	MC	Depth	% LC/MEM/DSP	%Eff
11	2	8	80 / 33 / 100	73
12	1	16	83 / 25 / 100	90

Table 5: k-NN Experiments. "Depth" indicates the number of *target descriptors* buffered per kernel, and "%Eff" indicates the percentage of peak performance achieved.

reuse and locality. The loop at line 3 iterates over all reference descriptors. The evaluation is based on finding k=2 nearest neighbors, a common scenario in vision algorithms, where a much further second nearest neighbor indicates a strong match. As in MMM, our compiler can generate kernel pipelines based on either loop unrolling or loop interchange (Section 4.2). We have found that in this application, interchanging loops (to support a large number of parallel kernel pipelines) is not cost effective due to excessive buffering requirements. Therefore, we focus on loop unrolling (lines 6-10) to create a large kernel, which is then replicated 8 times for parallel processing. We target the Altera DE4 platform for this evaluation, and can make a direct comparison against a high-quality hand designed implementation that we are using in an internal research project.

Table 5 presents our experimental configurations and results. Row **11** reports an implementation that uses two memory controllers. We were only able to successfully implement on-chip buffers that were 8 *target descriptors* deep, and consequently achieved only 73% of the potential throughput. The implementation reported in Row **12** has removed one memory controller, which has freed enough resources to double the buffer size. This halved the number of times that the "ri" loop (line 4) was executed, which effectively halves the bandwidth requirements. The result is that the implementation in Row **12** is able to reach 90% of the unrolled kernel pipeline's peak throughput. In comparison, our hand designed implementation can pack in 16 kernels on the EP4SGX530 on the Altera DE4, and achieve twice the performance over our compiler-generated implementation.

5.3 Two Dimensional Convolution

Listing 5 shows our final example, a direct two-dimensional convolution. In our evaluations (using a 2048-by-2048 data set, and 32-by-32 blocks), we focus on small convolution sizes (5-by-5 and 13-by-13) because larger convolutions are

[4]This example has a non-perfect loop nest due to the reduction step (implemented via macro StoreMin on line 12). Our compiler instantiates special hardware for known reductions. The rest of the loop nest is compiled as described.

```
1 void calc_2d_filter(float *IN,
    float *OUT, float *FILTER, unsigned Sz,
    unsigned BSz, unsigned FSize) {
2   unsigned starti,startj,i,j,ii,jj;
3   for (starti=0;starti<Sz;starti+=BSz)
4    for(startj=0;startj<Sz;startj+=BSz)
5     for(i=starti;i<starti+BSz;i++)
6      for(j=startj;j<startj+BSz;j++)
7       for(ii=0;ii<FSz;ii++)
8        for(jj=0;jj<FSz;jj++)
9         OUT[i*Sz+j]+=IN[(i+ii)*(Sz+FSz-1)
            +j+jj]*FILTER[ii*FSz+jj];
```

Listing 5: Blocked Two Dimensional Convolution.

#	Filter Size	% LC/MEM/DSP	GFlop/sec (%)
13	5	58 / 24 / 22	4.4 (69)
14	13	60 / 26/ 22	6.398 (99.9)

Table 6: 2D Convolution Experiments.

more commonly implemented with the help of Fast Fourier Transforms. As with the previous examples, the loop-based implementation has been blocked for data locality.

In this example, the compiler applies the loop interchange optimization to the two inner-most loops (which were causing a loop-carried dependency in the loop body), and moves them above the two loops originally at lines 5 and 6. When processing the IN buffer, the compiler detects two sequential accesses due to loop variables j and jj, with a computable access pattern between them and outside them due to loop variables i and ii. The compiler uses this information to infer a two dimensional sliding window buffer, which is instantiated along with a buffer to store the entirety of FILTER, which has a very significant effect on performance.

Our results are shown in Table 6. The compiler generated design fits 32 kernel pipelines onto the LX240T of the ML605. The system with 5-by-5 filters achieves 4.4 GFlop/sec, or 69% of the theoretical peak throughput on 5-by-5 filters. For 13-by-13 filters, increased data reuse allows nearly 100% of the kernel pipeline's theoretical peak throughput (6.4GFlop/sec) to be reached.

As a reference, Bao, et al. [4] (also compared to for MMM in Section 5.1.1) report a hand-tuned implementation of a 2048-by-2048 two-dimensional convolution with a filter size of 5-by-5. Their design, like ours, contains 32 processing elements. However, they only report 2.04 GFlop/sec of sustained performance. We suspect their lower performance is due to differences in the exploitation of data reuse.

6. CONCLUSIONS AND FUTURE WORK

We have presented a compiler that creates full FPGA implementations of perfect loop nests directly from software source code. Three test programs and two real world platforms demonstrate the efficacy of the compiler, and achieve performance ranging from a factor of 4 slower to a factor of 2 faster than hand designed implementations. This demonstrates that our compiler is a promising step towards enabling FPGA design creation by software developers without hardware design skills.

We are considering several enhancements to our compiler:

- Allowing imperfect loop nests to support more programs.
- Including polyhedral analyses to support better optimizations and allow the compiler to create a blocked computation structure.

• Targeting platforms that contain multiple FPGAs and very different memory interfaces, such as Convey's[3].

Acknowledgements: Funding for this work was provided in part by NSF CCF-1012851 and by Altera. We thank Eric Chung, Michael Papamichael, and Yu Wang for technical assistance, the individuals that assisted with proofreading, and the anonymous reviewers. We thank Xilinx and Altera for their FPGA and tool donations. We thank Bluespec for their tool donations and support.

7. REFERENCES

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 25–28, New York, NY, USA, 2011. ACM.

[2] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. FPGA-Specific Synthesis of Loop-Nests With Pipelined Computational Cores. *Microprocessors and Microsystems*, June 2012.

[3] J.D. Bakos. High-Performance Heterogeneous Computing with the Convey HC-1. *Computing in Science Engineering*, 12(6):80 –87, nov.-dec. 2010.

[4] Wenqi Bao, Jiang Jiang, Yuzhuo Fu, and Qing Sun. A Reconfigurable Macro-Pipelined Systolic Accelerator Architecture. In *FPT*, pages 1–6, 2011.

[5] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Samuel Bayliss and George A. Constantinides. Optimizing SDRAM bandwidth for Custom FPGA Loop Accelerators. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 195–204, New York, NY, USA, 2012. ACM.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[9] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP '08, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[11] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing : Does the Future Include Custom Logic , FPGAs , and GPGPUs? *International Symposium on Microarchitecture (MICRO-43), Atlanta, GA, 2010*, pages 225–236, 2010.

[12] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and evaluation of the CoRAM Memory Architecture for FPGA-Based Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, New York, NY, USA, 2012. ACM.

[13] Jason Cong, Peng Zhang, and Yi Zou. Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 185–192, Piscataway, NJ, USA, 2011. IEEE Press.

[14] Pedro C. Diniz and Joonseok Park. Data Reorganization Engines for the Next Generation of System-on-a-Chip FPGAs. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, FPGA '02, pages 237–244, New York, NY, USA, 2002. ACM.

[15] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56, New York, NY, USA, 2012. ACM.

[16] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - Polyhedral Optimization in LLVM . In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[17] Robert Kirchgessner, Greg Stitt, Alan George, and Herman Lam. VirtualRC: A Virtual FPGA Platform for Applications and Tools Portability. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 205–208, New York, NY, USA, 2012. ACM.

[18] Christian Lengauer. Loop parallelization in the polytope model. In *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 398–416, London, UK, UK, 1993. Springer-Verlag.

[19] Michael K. Papamichael and James C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing NOCS in the Context of FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

[20] David Barrie Thomas, Lee Howes, and Wayne Luk. A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 63–72, New York, NY, USA, 2009. ACM.

[21] Jingling Xue. On Loop Restructuring by Converting Imperfect to Perfect Loop Nests. In *IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, 1996. ICAPP '96.*, pages 421 –429, jun 1996.

[22] `www.ece.cmu.edu/~coram`.

[23] `www.jacquardcomputing.com/roccc/`.

[24] `http:/www.llvm.org/`.

[25] `www.impulseaccelerated.com/products.htm`.

[26] `www.mentor.com/products/fpga/handel-c/`.

[27] `www.mentor.com/esl/catapult/overview`.

[28] `www.xilinx.com/products/design-tools/vivado/index.htm`.

[29] `www.altera.com/devices/processor/nios2/tools/c2h/ni2-c2h.html`.

[30] `www.xilinx.com/support/documentation/data_sheets/ds150.pdf`.

[31] `www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf`.

[32] `www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.